Program Design & Data Structures (Course 1DL201)
Uppsala University Autumn 2021/Spring 2022
Homework Assignment 1: Computational Linguistics

Prepared by Dave Clarke and Johannes Borgström

Lab: Thursday 18 November and Monday 22 November
Submission Deadline: 18:00, Wednesday 1 December 2021

Computational Linguistics

Computational Linguistics is the discipline that studies natural language from a computational perspective by using computational models to analyse vast volumes of text in order to understand how language is used, among other things. Corpus Linguistics is one particular branch of Computational Linguistics that heavily uses computational techniques to understand the meaning of words from the contexts in which they are used. Specifically, Corpus Linguistics is interested in *collocations*, sequences of words that appear together more often than one would expect by chance. For example, one would expect to see *strong tea* in texts more often than *powerful tea*, yet *powerful computer* would appear more often than *strong computer*—ultimately stating something interesting to Computational Linguists about the difference in meaning of *strong* and *powerful*. Two important special cases are sequences of words that often appear at the beginning or end of sentences.

Goal

The goal of this assignment is to write a few Haskell functions for a computational linguistics toolbox.

The following are the core data types to be used in your program:[1]

```
type Sentence = [String]
type Document = [Sentence]
type WordTally = [(String, Int)]
type Pairs = [(String, String)]
type PairsTally = [((String, String), Int)]
```

The type `Sentence` represents sentences as lists of words—one can assume that raw documents have been preprocessed and the words have been extracted. The type `Sentence` represents documents as lists of sentences. The `WordTally` data type can be used to record word counts. The data type `Pairs` is used for lists of words that are *adjacent* to each other in the text. Finally, `PairsTally` is similar to `WordTally` except that it is used to record the number of occurrences of pairs of words.

The functions to be implemented (described in the next section) take a document and compute word counts, pairs of words that occur together, and simple statistics about those. The first couple of chapters of Jane Austin's *Pride and Prejudice* are available (variable `austin` in module `PandP`) to experiment with.

[1] The keyword `type` introduces a synonym for an existing type. For instance, the declaration `type Sentence = [String]` allows the programmer to use `Sentence` interchangeably with `[String]`.

Work to be Done

Download the files `PandP.hs` and `CompLing.hs` from Studium: `PandP.hs` contains preprocessed text from *Pride and Prejudice*, and `CompLing.hs` is for your solution. `CompLing.hs` also includes a number of tests to help develop your implementation.

You are required to implement the following functions (empty implementations are provided in the file `CompLing.hs`):

1. A function `wordCount :: Document -> WordTally` that computes a tally of all the distinct words appearing in the document. For example, the text `"A rose is a rose. But so is a rose."` is encoded in Haskell as the list of sentences `[["a", "rose", "is", "a", "rose"],["but", "so", "is", "a", "rose"]]`. The tally for this document is:

   - a — 3
   - rose — 3
   - is — 2
   - but — 1
   - so — 1

   This result could be represented in Haskell as follows, though the order of elements in the list is not specified:

   ```
   [("rose", 3), ("a", 3), ("is", 2), ("but", 1), ("so", 1)]
   ```

2. A function `adjacentPairs :: Document -> Pairs` that yields a list of all adjacent pairs of words appearing in the document, with duplicates present.

   For instance,

   ```
   adjacentPairs [["time", "for", "a", "break"], ["not", "for", "a", "while"]]
       == [("time","for"),("for","a"),("a","break"),("not","for"),("for","a"),("a","while")]
   ```

3. Two functions `initialPairs :: Document -> Pairs` and `finalPairs :: Document -> Pairs` that return a list of all pairs of words appearing at the start (or end) of sentences in the document, with duplicates present.

   For instance,

   ```
   initialPairs [["time", "for", "a", "break"], ["not", "yet"]]
       == [("time","for"),("not", "yet")]
   finalPairs [["time", "for", "a", "break"], ["not", "yet"]]
       == [("a","break"),("not", "yet")]
   ```

4. A function `pairsCount :: Pairs -> PairsTally` that computes a tally of all pairs, such as those computed by `adjacentPairs`.

   For instance, `pairsCount [("big","bear"),("bear","big"),("big","dog")]` would result in the tally:

   - `big, bear` — 2
   - `big, dog` — 1

Note that here, we do not care about the order of words in a pair. For instance, we consider `("big","bear")` and `("bear","big")` to both represent the same pair of words, so they must not both appear in the tally.

How this tally is represented in Haskell is a design decision, meaning that either:

```
[(("bear","big"),2), (("big","dog"),1)]
```

or

```
[(("big","bear"),2), (("big","dog"),1)]
```

or any reordering of these, are all valid ways of representing the tally.

5. A function `neighbours :: PairsTally -> String -> WordTally` that takes a tally of pairs, such as computed by the `pairsCount` function, and a word and gives all the words that appear with that word in the tally of pairs along with the number of occurrences.

   For instance,

   ```
   neighbours [(("bear","big"),2),(("big","dog"),1)] "big"
   ```

   should return `[("bear",2),("dog",1)]` or some reordering of this list.

6. A function `mostCommonNeighbour :: PairsTally -> String -> Maybe String` returns the word that occurs most frequently with a given word, based on a tally of pairs. The `Maybe` data type is used to represent the result:

   - If the word does not appear in the tally, the result is `Nothing`
   - If the word appears in the tally, and the neighbour with the largest count is `"foo"`, then the result is `Just "foo"`. If more than one word appears in neighbours with equal highest count, then the result is `Just x`, where `x` is one of those words.

   The `Maybe` data type is described in the next section.

Data types

You will be using the standard `String` datatype, and its default comparison functions. You do not need to consider issues such as capitalization, diacritics, or different representations of the same glyph: you may assume that preprocessing has taken care of these issues.

The `Maybe` data type

The `Maybe` data type is used for the result of functions that do not always return a valid result. It is declared in the standard library as

```
data Maybe a = Just a | Nothing
```

A valid result is indicated as `Just x`, for some value `x`, and no valid result is indicated as `Nothing`.

Datatype libraries

The required functions can be written using only functions and datatypes from the standard Haskell prelude. However, you may use other functions and datatypes from the standard `Data.*` libraries—but be aware that increased complexity increases the risk of bugs!

Running and Testing

There are (at least) two ways of loading your program into **ghci**, assuming it is stored in the directory *dir*:

- Run `ghci dir/CompLing.hs` in a shell.
- Type `ghci` in a shell. Then go to the directory where your file is stored (using the command `:cd dir`), and issue the command `:l CompLing.hs`.

After modifying and saving your code in an editor, enter `:r` within **ghci** to reload the file.

Ten test cases have been provided for you in the file `CompLing.hs`. These can be run by entering `runtests` in **ghci**. Feel free to add further test cases to test your code more thoroughly.

Grading

Your solution is scored on a U/6-10 scale based on two components: (1) functional correctness and (2) style and comments.

1. Functional correctness:

Your program will be run on an unspecified number of grading test cases that satisfy all preconditions but also check boundary conditions. Each test case is based on queries to some (possibly empty) document using the functions provided.

We reserve the right to run these tests automatically, so **be careful to match exactly the imposed file name, function names, and argument orders**—that is, don't change what we've provided for you.

Advice: Run your code in a freshly started Haskell session before you submit, so that declarations that you may have made manually do not interfere when you test the code.

The score for this component is computed as follows:

- If your solution was submitted by the deadline, your file `CompLing.hs` loads in `ghci`, and it fails at most one of the *test cases provided* in `CompLing.hs`, and it passes 70% of the *grading test cases*, you get (at least) a 3 for functional correctness. Otherwise (including when no solution was submitted by the deadline), you do not pass the homework assignment (U).
- If your program passes all of the provided test cases and at least 85% of the grading test cases, you get (at least) a 4 for functional correctness.
- If your program passes all test cases, you get a 5 for functional correctness.

2. Style, specification, and readability:

Your program is scored for style and comments according to our *Coding Convention*. The following criteria will be used:

- suitable breakdown of your solution into auxiliary/helper functions;
- function specifications, statements of purpose, and variants (where needed);
- code readability and indentation;
- sensible naming conventions followed.

You do not need to provide datatype representation conventions and invariants.
The score for this component is computed as follows:

- If your program's style and comments are deemed a serious attempt at following these criteria, and there are at most two significant errors or omissions in each function comment, you get (at least) a 3 for style and comments. Otherwise, you do not pass the homework assignment (U).
- If you have largely followed these criteria, with very few significant omissions or errors, you get a 4 for style and comments.
- If you have followed these criteria with at most minor omissions or oversights, you get a 5 for style and comments.

A significant error or omission is one that directly contradicts the coding conventions, or would cause a user of your library to pick the wrong function, or cause code using the function in question to compute the wrong result, or not terminate, for some inputs.

Final Score

Component scores are converted to a pass/fail outcome and a final score as follows:

1. If either component (functional correctness, and style and comments) has a U score, you fail the assignment.
2. Otherwise, your final score is the sum of the two component scores.

Modalities

- The assignment will be conducted in pairs of 2 (or possibly 3). Pairs have been assigned via Studium. *If you have not been matched, contact Eva as soon as possible. If you cannot find your partner until Thursday 18 November, please contact Eva to assign you a new partner—if possible.*
- Your solutions should consist of one file only, named `CompLing.hs`, and be submitted via Studium. Only one solution per group is to be submitted. Ensure that **both** team members' names appear in the submitted file.

By submitting a solution you are certifying that it is solely the work of your group, except where explicitly attributed otherwise.

Good luck!